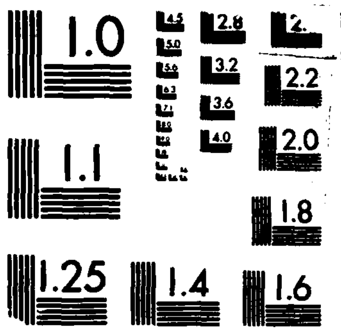
AD-A193 645 PRACTICAL HIGHER-ORDER FUNCTIONAL AND LOGIC PROGRAMMING BASED ON LAMBDA-C..(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE F S SILBERMANN ET AL. 1/1

UNCLASSIFIED JAN 88 TR88-004 N00014-86-K-0680 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

Practical Higher-Order Functional and Logic Programming Based on Lambda-Calculus and Set-Abstraction

TR88-004

January 1988

Frank S.K. Silbermann and Bharat Jayaraman

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A

Practical Higher-order Functional and Logic Programming based on Lambda Calculus and Set Abstraction[†]

(Summary)

Frank S.K. Silbermann
Bharat Jayaraman

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

[illegible]

Abstract

We propose new variation of relative set abstraction as an extension to a lambda-calculus based functional language. This feature interacts orthogonally with the standard functional language capabilities, yet provides the full expressive power of first-order Horn-logic programming, as well as a very useful subset of higher-order Horn-logic programming. This resulting language lends itself to efficient interpretation, in that complete operational procedures are possible without computationally expensive primitives such as higher-order unification, unification relative to an equational theory, or general theorem-proving.

† This research is supported by grant DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

1. Introduction

From the perspective of predicate-logic programming, functional programming offers three important additional capabilities: infinite data objects, higher-order objects, and directional (non-backtrackable) execution. From the perspective of functional programming, the unique capabilities of predicate-logic programming are its support for constraint reasoning, via unification over first-order terms, and flexible execution moding (non-directionality). We describe in this paper an approach that combines these capabilities in a single language. The language is efficient in that functional programming can be carried out without backtracking, and its subset of higher-order logic programming can be carried out without potentially expensive operations such as unification relative to an equational theory [GM84], general theorem-proving [MMW84], or higher-order unification [MN86, R86]. It is elegant in that all language constructs combine in an orthogonal way, simplifying the denotational semantics. It is declarative in that the language does not depend upon imperative control constructs or destructive assignments, and the language is extensional, i.e., two semantically identical objects can be used interchangeably in all contexts. Existing approaches fall short in that they either

(a) support no higher-order programming at all [GM84, DP85, YS86], or

(b) require higher-order unification [MN86, R86] (in general undecidable), or

(c) have *no extensional declarative semantics* [SP85, W83], *or*

(d) have no identifiable purely functional subset [R85, L85].

The importance of our proposed work is that it overcomes all of the above shortcomings in a simple way. Our approach is to enhance a lambda-calculus based functional language with a new variation of relative (not absolute) set abstraction. With this set-abstraction mechanism, our language subsumes all of first-order Horn logic programming, and much of higher-order Horn logic as well.

We believe that functional programming is a better basis for a unified declarative language than Horn-logic, because propagation of higher-order objects does not require higher-order unification. It is also amenable to a correct and efficient implementation, as recent compilation techniques have shown [P87]. Because all logic programming is encapsulated within set-abstraction, it is possible to syntactically identify those parts of a program that require backtracking and those parts that do not. With this distinction, purely functional computations may be performed more efficiently.

We should point out two significant differences between our approach and similar work:

(i) Although our relative set-abstraction is *syntactically* similar to Turner's set construct in Miranda [T85], it is semantically very different. Turner's construct is essentially a high-level notation for defining lists. No semantic power is added by this construct; on the contrary, representing a set as a list introduces an undesirable element of nondeterminism in his language. (A function on sets may produce different results depending upon the list representation used.) Our language provides true (recursively-enumerable) sets.

(ii) Our seta are closer in spirit to Darlington's absolute set abstraction [DFP86], but with two important differences. All variables introduced in Darlington's set abstraction are implicitly quantified over the set of first-order terms—relaxation of this restriction requires higher-order unification, and even then many legal programs are unexecutable specifications. We permit the variables to be quantified over any recursively-enumerable set, which must first be explicitly specified. Hence our use of the term 'relative'. Variables in our relative set abstraction can range over a set of functions, a set of sets, etc. Second, Darlington does not specify how the sets he defines interact with other language constructs. Our approach is orthogonal in that it permits set-valued functions, lists of sets, etc.

To support infinite objects (both infinite terms and infinite sets), the operational semantics is based on the usual normal-order evaluation. Within a relative set-abstraction, we defer as long as possible the enumeration of elements from the generator set of first-order terms, reducing conditions over first-order terms via a simplified form of narrowing, which we described in an earlier paper [JS86]. To maintain extensionality, equality over higher-order types (sets and functions) is undefined.

The rest of this summary divides into the following sections: section 2 presents language features and some examples to show the versatility of the constructs for higher-order functional and logic programming; section 3 presents a declarative and procedural semantics of the language, along with correctness theorems; and section 4 presents conclusions.

2. Language Framework

2.1 Informal Description

We describe below a language called Setλ. The data values of Setλ consist of functions, terms, and sets:

1. Functions are defined by λ-abstraction. The primitive functions are the usual LISP-like primitives **car**, **cdr**, **atom**, etc. [M65].

2. Terms are built up from the primitive constructor **cons**, and may contain atoms, functions, or sets. A term may be infinite. First-order terms are built up from **cons** and atoms only, and are of finite size.

3. Sets are defined by set abstraction. The primitive sets are the set of atoms (A) and the set of first-order terms (T). Sets may also contain terms, functions and other sets.

As in LISP, lists are a subset of terms, but written in the `[...]` notation, e.g. `['apple, 'orange, 'grape]`. For sake of brevity, we omit discussion of integers and their operations in this presentation, although they will be include in the full paper.

At the top-level, a Setλ program usually has the form

`letrec` *name* `be` *expr*, ..., *name* `be` *expr* `in` *expr*

where an expression *expr* may be an identifier, data value or any of the following:

1. letrec: as defined above
2. primitive: `car`(*expr*), `cdr`(*expr*), etc.
3. conditional: `if` *condition* `then` *expr* `else` *expr*
4. λ-abstraction: λ *vars* . *expr*
5. application: *expr*(*expr*, ..., *expr*)
6. set-abstraction: { *set-clause* ; ... ; *set-clause* }

A *condition* has the form

$expr_1 = expr_2$ or $expr_1 \neq expr_2$, or `atom?`(*expr*), or `pair?`(*expr*), etc.,

and a *set-clause* has the form

expr : *enumerations* , *conditions*

where *enumerations* specifies the generator sets for n distinct variables as follows:

$var_1 \in expr_1, \ldots, var_n \in expr_n$

and *conditions* has the form:

condition, ..., *condition*

Notes: (1) Within a set-clause, a variable var_i defined by enumeration $var_i \in expr_i$, may be used in defining generator sets of *later* enumerations, as well as in the conditions and head expression of the set-clause, but may not be used in $expr_i$ or earlier enumerations. The scope of a variable introduced in *enumerations* does not extend to other set-clauses in the set-abstraction.

(2) Because we use relative set-abstraction, every newly-introduced variable in a set-clause has a corresponding generator-set. The set denoted by a set-abstraction is the union of the sets denoted by each of its set-clauses.

(3) Equality and inequality are defined only between two first-order terms. It is a type-error to use these two operations between sets or functions. When used between two infinite terms made up from atoms and cons, the result is $\perp$ if the terms are equal and `false` otherwise.

(4) We do *not* permit a non-membership condition of the form *var* $\notin$ *expr*. This operation is disallowed because sets are not assumed to recursive; disallowing this operation also avoids the possibility of paradoxical sets.

2.2 Examples

Functional Programming

```
let
   append be λ x y .  if null(x) then y else cons(car(x), append(cdr(x), y))
   map be λ f . λ l .  if null(l) then [] else cons(f(car(l)), map(f, cdr(l)))
   infinite be cons('a, infinite)
in
   ...
```

First-order and higher-order functions, as well as infinite objects can be defined in the usual manner. Note that functions can be expressed in curried form, as the map example illustrates.

Logic Programming

```
let
    split be λlist . {cons(x,y)  :  x ∈ T,  y ∈ T,  append(x,y) = list}
in
   ...
```

Note: (1) T is the set of finite first-order terms. The enumerations x ∈ T, y ∈ T are needed because the set-abstraction is relative. Operationally, the generation of elements from T is always delayed as much as possible. Thus, the condition append(x,y) = list will be first narrowed to obtain bindings for x and y; then the membership of x and y in T will be verified (trivially).

(2) An operation such as append, which is used inside a set-abstraction as well as outside it, might be compiled in two different ways corresponding to these two uses.

Set Operations

```
let
    crossprod be λ s1 s2 .  {cons(x,y)  :  x ∈ s1,  y ∈ s2}
    filter be λ p s .  {x  :  x ∈ s,  p(x) = true}
    union be λ s1 s2 .  {x  :  x ∈ s1;  x  :  x ∈ s2}
in
   ...
```

The operations crossprod and filter are similar to those in Miranda [T85]. The union example illustrates the use of multiple clauses. Note that the occurrences of x in the two clauses are independent.

Because nondeterministic enumeration is the only primitive operation on sets, it is unnecessary to remove duplicates in the construction of sets. Note that one cannot define an operation to compute the size of a set in Setλ, because such an operation would be analogous to Prolog's meta-logical features.

Higher-order Functional and Horn-logic programming

```
let
    one be λ v . 'a
    two be λ v . 'b
    three be λ v . 'c
in
   {f : f ∈ {one,two,three}, map(f)(['x, 'y, 'z]) = ['c, 'c, 'c]}
```

The result of the above set-abstraction is the set {three}. In this example, the generator set for f, {one, two, three}, is first enumerated to obtain a function which is then passed on to map. Those that satisfy the equality condition are kept in the resulting set.

We close this section by showing how any Horn-logic program can be mechanically converted into Setλ. Consider the following program, written in Prolog syntax [WPP77], which has a unit clause, a conditional clause, and a top-level goal:

```
rev([], []).
rev([H|T], Z) :- rev(T,Y), app(Y, [H], Z).
? rev(L, [a, b, c]).
```

The converted Setλ program would be as follows:

```
let
   rev be { [[], []] : true ;
            [cons(h, t), z] : h,t,y,z ∈ T, v1 ∈ rev, v2 ∈ app,
                              v1 = [t, y], v2 = [y, [h], z]}
in
   { l : l ∈ T, v1 ∈ rev, v1 = [l, ['a, 'b, 'c]] }
```

In Horn-logic, every predicate implicitly defines a set of terms—the set of argument term-tuples for which the predicate is true. We have taken the liberty of writing h,t,y,z ∈ T instead of four separate enumerations.

3. Semantics

3.1 Declarative Semantics

Though Setλ supports logic programming, it is essentially a functional programming language. We therefore present its semantics in the style conventional for functional languages [S77]. The domains are as follows, where $A_\perp$ is the flat domain of atoms, P is the non-flat domain of terms, F is the domain of functions, and S is the domain of sets:

$$D = A_\perp + T_\perp + P + F + S$$

$$P = D \times D$$

$$F = [D \mapsto D]$$

$$S = \mathcal{P}(D)$$

where $\mathcal{P}(D)$ stands for the power-set of D. Space precludes us from presenting the details of the structure of this power-domain in this summary; these will be given in the full paper. We should note that its construction is similar to the power-domain for "angelic" nondeterminism, as described by Broy [B85].

In the definitions below, the semantic function $\mathcal{E}$ maps general expressions to denotable values, and $\mathcal{S}$ handles the specific case for set-abstractions. The environment, ρ, maps identifiers to denotable values, and belongs to the domain $[Id \mapsto D]$. In this summary we provide only the semantic equations for (relative) set-abstractions, the novel part of the language. The equations of $\mathcal{E}$ for the other five forms of expressions defined in section 2.1 are the conventional ones for a lazy functional language. We do not present these details here, but will in the full paper.

$$\mathcal{E}(\llbracket\{setclause_1; \ldots; setclause_n\}\rrbracket,\ \rho) = \mathcal{S}(\llbracket\{setclause_1; \ldots; setclause_n\}\rrbracket,\ \rho)$$

$$\mathcal{S}(\llbracket\{setclause_1; \ldots; setclause_n\}\rrbracket,\ \rho) = \cup_{i=1,n}\mathcal{S}(\llbracket\{setclause_i\}\rrbracket,\ \rho)$$

$$\mathcal{S}(\llbracket\{expr\ :\ var \in expr_2, enumerations, conditions\}\rrbracket,\ \rho)$$
$$= \cup_{g\in\mathcal{E}(\llbracket expr_2\rrbracket,\ \rho)}\mathcal{S}(\llbracket\{expr\ :\ enumerations, conditions\}\rrbracket,\ \rho[var \leftarrow g])$$

$$\mathcal{S}(\llbracket\{expr\ :\ condition, conditions\}\rrbracket,\ \rho)$$
$$= \textbf{if } \mathcal{E}(\llbracket condition\rrbracket,\ \rho) \textbf{ then } \mathcal{S}(\llbracket\{expr\ :\ conditions\}\rrbracket,\ \rho) \textbf{ else } \phi$$

$$\mathcal{S}(\llbracket\{expr\}\rrbracket,\ \rho) = \{\mathcal{E}(\llbracket expr\rrbracket,\ \rho)\}$$

The functions $\mathcal{E}$ and $\mathcal{S}$ are mutually recursive. Their meaning is the least fixed point of the recursive definition. Operations are continuous because all sets are recursively-enumerable and the union operation $\cup$ is continuous.

3.2 Operational Semantics

In the absence of set-abstraction, Setλ is executed using normal-order evaluation like any other lambda-calculus based functional language. When evaluating a set-abstraction, the interpreter must be capable of producing any element in the denoted set. This is sufficient, because a set is examined only in enumeration expressions.

A simple non-deterministic operational procedure based on pure reduction can be built directly from the denotational semantics. Everywhere the declarative semantics equates a set-abstraction to a union of simpler set-expression, the operational semantics non-deterministically rewrites the set-abstraction to anyone of them, until a singleton set results. However, such a generate-and-test approach is needlessly inefficient. When the generator set is the set of atoms (A), the set of finite first-order terms (T), we defer the enumeration, and depend on conditions such as equality constraints or LISP-like predicates (like `pair?`, etc.) to narrow the choice of plauseable candidates. So while enumerations of first-order terms are delayed, partial evaluation of conditions where possible is given priority.

Because we are dealing with an expression-based language, our handling of conditions such as equality resembles narrowing in term-rewriting systems [GM84, DP85, YS86]. In contrast to classical narrowing, we will need to perform reductions only at the outermost level, and we need narrow only first-order terms, rather than arbitrary values. This avoids the explosive growth possible in more general narrowing-based computations. Our operational procedure comes closest to Reddy's lazy narrowing [R85]. The main difference is that we do not need a primitive unification at each narrowing step, as function application uses only one-way substitution. Unification is diffused into the lazy narrowing process, sketched below in the form of reduction rules.

We gave a detailed account of this process in our earlier paper [JS86]. In this summary, we sketch some of the reduction rules for equality, ignoring environments and "variable capture" during function application; the full paper provides a more thorough treatment. We assume the set-abstraction has one clause for simplicity. The notation $S[cond]$ means that $cond$ is a condition appearing in set-abstraction S; the notation $S[enums; conds]$ means that the enumerations $enums$ and conditions $conds$ together appear in S. The notation $S[x;\ y] \rightarrow S[p;\ q]$ means that the enumerations x in S are replaced by p, and the conditions y in S are replaced by q. Finally, the notation $expr[e_1]$ means that e_1 is the outermost reducible expression of $expr$. For example, if $expr$ were `car(f(x))`, e_1 would be `f(x)`.

1. *Application*

$$S[expr[\mathtt{f}(e_1,\ldots,e_n)] = expr_2] \quad \rightarrow \quad S[expr[expr_1\ \sigma] = expr_2]$$

where $\mathtt{f}$ is $\lambda\ v_1 \ldots v_n\ .\ expr_1$ from the associated environment, and

$\sigma = \{v_1 \leftarrow e_1, \ldots, v_n \leftarrow e_n\}$.

2. *Decomposition*

$$S[\mathtt{cons}(e_1, e_2) = \mathtt{cons}(e_3, e_4)] \quad \rightarrow \quad S[e_1 = e_3,\ e_2 = e_4].$$

3. *Primitives*

$$S[x \in \mathtt{T};\ expr[\mathtt{null}(x)] = expr_2] \quad \rightarrow \quad (S[\mathtt{true};\ expr[\mathtt{true}] = expr_2])\ \rho$$

where $\rho = \{x \leftarrow \mathtt{[]}\}$.

$$S[x \in \mathtt{T};\ expr[\mathtt{null}(x)] = expr_2] \quad \rightarrow \quad (S[x_1 \in \mathtt{T},\ x_2 \in \mathtt{T};\ expr[\mathtt{false}] = expr_2])\ \rho$$

where $\rho = \{x \leftarrow \mathtt{cons}(x_1, x_2)\}$.

4. *Variable Binding*

$$S[x \in \mathtt{T};\ x = \mathtt{cons}(expr_1, expr_2)] \quad \rightarrow \quad (S[x_1 \in \mathtt{T},\ x_2 \in \mathtt{T};\ x_1 = expr_1,\ x_2 = expr_2])\ \rho$$

where $\rho = \{x \leftarrow \mathtt{cons}(x_1, x_2)\}$.

Note that there are two reduction rules for primitive null, in case 3. (Appropriate rules can be similarly written for other primitives.) This causes nondeterministic branching in the computation of sets. A successful derivation from a set-abstraction S is one which eventually terminates in a singleton set, say s. Notationally, this is expressed as $S \rightarrow^* s$. A derivation is said to fail if some condition is determined to be false. The following two theorems will establish the correctness of the operational semantics.

Soundness Theorem:

Given a set-abstraction S and a Setλ program $\mathcal{P}$, and a derivation $S \rightarrow^* s$, it follows that $s \subseteq T$, where T is the set denoted by S according to the declarative semantics.

Completeness Theorem:

Given a set-abstraction S and a Setλ program $\mathcal{P}$, and a singleton set $s \subseteq T$, where T is the set denoted by S according to the declarative semantics, there exists a derivation $S \rightarrow^* s$.

A more rigorous statement of the theorems and their proofs are given in the full paper.

4. Conclusions

During the past decade, the integration of functional and logic programming has been a topic of great interest [BL86]. Most of the efforts (including our earlier work [JS86]) have dealt only with first-order functional and logic programming. Those efforts that do support higher-order functional programming (see section 1.0) do not provide *both* a clear declarative semantics and an efficient operational semantics. We believe we have overcome both these shortcomings in this paper.

Because our approach is based on an extensional higher-order functional lanaguage, in which higher-order objects may not be compared, we avoid the need for higher-order unification. We use relative set-abstraction in a novel way to obtain first-order Horn-logic programming as well as an extensional subset of higher-order logic programming. The declarative semantics of these constructs is based on well-developed methods [S77, B85]; and the operational semantics is based on results from our earlier paper [JS86], which has close connections with narrowing. The proofs of the correctness theorems are tedious, but not technically difficult.

References

[B85] M. Broy, "Extensional Behavior of Concurrent, Nondeterministic, and Communicating Systems," In *Control-flow and Data-flow Concepts of Distributed Programming*, Springer-Verlag, 1985, pp. 229-276.

[BL86] M. Bellia and G. Levi, "The Relation between Logic and Functional Languages: A Survey," In *J. of Logic Programming*, vol. 3, pp.217-236, 1986.

[DP85] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming*, Boston, MA, July 1985, pp. 54–66.

[DFP86] J. Darlington, A.J. Field, and H. Pull, "Unification of Functional and Logic Languages," In DeGroot and Lindstrom (eds.), *Logic Programming, Relations, Functions and Equations*, pp. 37-70, Prentice-Hall, 1986.

[GM84] J. A. Goguen and J. Meseguer, "Equality, Types, Modules, and (Why Not?) Generics for Logic Programming," *J. Logic Prog.*, Vol. 2, pp. 179–210, 1984.

[JS86] B. Jayaraman and F.S.K. Silbermann, "Equations, Sets, and Reduction Semantics for Functional and Logic Programming," In *1986 ACM Conf. on LISP and Functional Programming*, Boston, MA, Aug. 1986, pp. 320-331.

[L85] G. Lindstrom, "Functional Programming and the Logical Variable," In *12th ACM Symp. on Princ. of Prog. Langs.*, New Orleans, LA, Jan. 1985, pp. 266–280.

[M65] J. McCarthy, et al, "LISP 1.5 Programmer's Manual," MIT Press, Cambridge, Mass., 1965.

[MMW84] Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: The Deductive-Tableau Programming Language," In *ACM Symp. on LISP and Functional Programming*, Austin, TX, Aug. 1984, pp. 323–330.

[MN86] D. Miller and G. Nadathur, "Higher-Order Logic Programming," In *Third International Conference on Logic Programming*, London, July 1986, 448-462.

[P87] S.L. Peyton Jones, "The Implementation of Functional Programming Languages," Prentice-Hall, 1987.

[R85] U. S. Reddy, "Narrowing as the Operational Semantics of Functional Languages," In *1985 Symp. on Logic Programming*, Boston, MA, July 1985, pp. 138–151.

[R86] J. A. Robinson, "The Future of Logic Programming," IFIP Proceedings, Ireland, 1986.

[S77] J. E. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory," MIT Press, Cambridge, Mass., 1977.

[SP85] G. Smolka and P. Panangaden, "A Higher-order Language with Unification and Multiple Results," Tech. Report TR 85-685, Cornell University, May 1985.

[T85] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types," in *Conf. on Functional Prog. Langs. and Comp. Arch.*, Nancy, France, Sep. 1985, pp. 1-16.

[WPP77] D. H. D. Warren, F. Pereira, and L. M. Pereira, "Prolog: the Language and Its Implementation Compared with LISP," *SIGPLAN Notices*, Vol 12., No. 8, pp. 109–115, 1977.

[W83] D. H. D. Warren, "Higher-order Extensions of Prolog: Are they needed?" Machine Intelligence 10, 1982, 441-454.

[YS86] J-H. You and P. A. Subrahmanyam, "Equational Logic Programming: an Extension to Equational Programming," In *13th ACM Symp. on Princ. of Prog. Langs.*, St. Petersburg, FL, 1986, pp. 209-218.

END
DATE
FILMED
8-88
DTIC